NOTES ON PARTIAL CORRECTNESS

by

Olaf Owe

Abstract. An attempt is done to make successful use of total functions within partial verification reasoning. It is shown how to obtain results with different degrees of "partialness". An application to (abstract) data types is suggested.

Research Report Series, Institute of Informatics,
University of Oslo, Norway
December 1977
Revised March 1978

This research is supported by The Norwegian Research Council for Science and The Humanities.

CONTENTS

1. Introduction

This paper presents some extensions of the approach to program verification given by Hoare [1,2]. We assume the reader is familiar with the Hoare system.

The ideas presented here are founded in the development of an interactive system for program verification, called PROVER [3], and are oriented towards mechanical verification.

The Hoare-system is based on predicate logic. A partial specification of a program†, S, is given by two formulas on the program variables, the precondition, P, and the postcondition, Q, written as

$$P \{S\} Q.$$

The meaning is: if P is valid just before an execution of S and if the execution terminates normally then Q holds immediately after the execution. Normal termination of a given execution is immediately decidable (from the run time messages).

We may say that a weak precondition and a strong postcondition makes a strong partial specification, thus

(1) $P\{S\}Q \Rightarrow P'\{S\}Q'$, if $P' \Rightarrow P$ and $Q \Rightarrow Q'$.

The strongest partial specification of S for a given postcondition is a fundamental concept.

In the Hoare system it is defined how to construct a precondition for each basic statement given an arbitrary postcondition, such that the partial specification generated is the strongest. For instance, the assignment statement has the strongest partial specification

$$Q_e^x \{x:=e\} Q .$$

Q_e^x is obtained by substituting the expression e for all free occurrences of the variable x in Q†† We shall allow x and e to be lists(i.e.vectors), the substitution is then simultaneous. This makes "right to left" verification convenient, i.e.given a postcondition of S a precondition of S is constructed by transformation over S from right to left.

---

† By a program we shall mean any statement (list) or body of a procedure declaration.

†† if necessary, renaming bound variables in Q so that no free variable in e is bound.

Any partial specification of a program that never terminates normally may be verified. Symbolically we will let "error" denote such a program. Hence, we may formulate the axiom

$$\text{true } \{\text{error}\} \text{ false}$$

which is the strongest possible partial specification.

2. Functionality.

It is useful according to Hoare[4], to interpret a program as a simultaneous assignment.

A program ,S, can be characterized by a unique partial function f_S, namely the result function of S on the state vector (i.e. a vector of all the program variables in S);and f_S is defined whenever S terminates normally.

Let $t_S(v)$ be the condition that S terminates normally with initial value v of the state vector. By definition we have

$$(2)\quad \forall v_1 \left(t_S(v_1) \Leftrightarrow \exists v_2 \; f_S(v_1) = v_2 \right)$$

The condition that S terminates normally with initial value v_1 and result value v_2 of the state vector, can now be defined:

$$RES_S(v_1,v_2) \equiv \underline{\text{if}} \; t_S(v_1) \; \underline{\text{then}} \; f_S(v_1)=v_2 \; \underline{\text{else}} \; \text{false} \; \underline{\text{fi}}$$

With this notion, the interpretation of P{S}Q can be specified as

$$P(v)\{S\}Q(v) \equiv$$

$$\forall v_1,v_2 \; (RES_S(v_1,v_2) \wedge P(v_1) \Rightarrow Q(v_2))$$

expressing that if S terminates normally and if P holds for the initial value of the state vector then Q holds for the result value.

In practical proving it is convenient to reduce the state vector as much as possible. Mechanically,two vectors of program variables can be determined : the input vector ,a, of S and the output vector ,x, of S ,such that we may regard t_S as a predicate of a and f_S as a function from values of a (input values) to values of x (output values). In general $\{a\} \cap \{x\} \neq \emptyset$, (i.e. some program variables may be changed by S and their initial values are not redundant).

A precondition is on input values and a postcondition is usually on output values, however, initial values of input/output variables may be accessed by use of auxiliary variables.

From the given interpretation we may now obtain a useful lemma:

Lemma 1

$$P\{S\}Q \Leftrightarrow \forall a\ t_s(a) \Rightarrow (P \Rightarrow Q^{x}_{f_s(a)})$$

Proof $P\{S\}Q \Leftrightarrow \forall a',x'(\ RES_s(a',x') \wedge P^{a}_{a'} \Rightarrow Q^{x}_{x'}\ )$, by interpretation where a' and x' are disjoint variable vectors, a' denoting input values, x' denoting output values.

$\Leftrightarrow \forall a',x'(t_s(a') \wedge x'=f_s(a') \wedge P^{a}_{a'} \Rightarrow Q^{x}_{x'})$, by definition of RES_s

$\Leftrightarrow \forall a',x'(\ t_s(a') \wedge x'=f_s(a') \wedge P^{a}_{a'} \Rightarrow Q^{x}_{f_s(a')})$, by simplification

$\Leftrightarrow \forall a' \quad (t_s(a') \wedge \exists x'(x'=f_s(a')) \wedge P^{a}_{a'} \Rightarrow Q^{x}_{f_s(a')})$, by simplification

$\Leftrightarrow \forall a' \quad (t_s(a') \wedge P^{a}_{a'} \Rightarrow Q^{x}_{f_s(a')})$, by (2)

$\Leftrightarrow \forall a \quad (t_s(a) \Rightarrow (P \Rightarrow Q^{x}_{f_s(a)})\ )$, by simplification. □

The rule of functionality given in [5] can be formulated in our notation as

Rule 1

$$\frac{P\{S\}Q}{\forall a\ t_s(a) \Rightarrow (P \Rightarrow Q^{x}_{f_s(a)})}$$

The conclusion is equivalent to the premise by the given lemma, and is called the partial function theorem of S with respect to $P\{S\}Q$, and expresses what is known about f_s apart form the fact that it exists. This rule may alternatively be formulated in this way, if a necessary condition for normal termination is wanted:

Rule 2

$$\frac{P\{S\}Q}{\forall a\ t_s(a) \Rightarrow (P \Rightarrow \exists x\ Q)}$$

This follows from (2). Rule 2 characterizes t_s, giving an upper bound for $\{a|t_s(a)\}$.

In [5] the rule of functionality introduced in Hoare [4] was shown to be inconsistent if used without restrictions. The restrictions suggested were based on interpretation of functions as partial functions. Every occurrence of a function had to be interpreted restrictively:

$\forall x\ P(f(x))$ is interpreted as $\forall y\ \forall x((x,y) \in f \Rightarrow P(y))$.

We let the restrictions be explicitly expressed by the predicate t_s, and will obtain partial specifications which have no restrictions.

We may look at the example given in Ashcroft et al [5]. Let S be x:=0; while true do null .
We may prove (using x=0 as invariant)

$$\text{true}\ \{S\}\ \text{false} \wedge x=0$$

Rule 1 gives (since x is the only input variable, and there are no input variables)

$$t_s \Rightarrow \left(\text{true} \Rightarrow (\text{false} \wedge x=0)^x_{f_s}\right)$$

By logical simplification we obtain

$$\neg\, t_s$$

Thus we have proved that S never terminates normally.

A partial function theorem may seem to be of little practical use since the truth value of the term $t_s(a)$ is usually unknown at (partial) verification time. However, the following lemma indicates that a partial theorem can be applied in any precondition of S without the restriction $t_s(a)$.

Lemma 2 $\qquad P\{S\}Q \Leftrightarrow (t_s(a)\wedge P)\{S\}Q \Leftrightarrow (t_s(a)\Rightarrow P)\{S\}Q$

This means that in a (partial) precondition the termination condition is redundant (and may be added or deleted).

Proof

i) $\quad P\{S\}Q$

$\Leftrightarrow (t_s(a) \Rightarrow (P \Rightarrow Q^x_{f_s(a)}))$, by lemma 1

$\Leftrightarrow (t_s(a)\Rightarrow((t_s(a)\wedge P)\Rightarrow Q^x_{f_s(a)}))$, tautology

$\Leftrightarrow (t_s(a)\wedge P)\{S\}Q$, by lemma 1

ii) $\quad (t_s(a)\Rightarrow P)\{S\}Q \Leftrightarrow (t_s(a)\wedge(t_s(a)\Rightarrow P))\{S\}Q$

$\Leftrightarrow (t_s(a)\wedge P)\{S\}Q$ □

The lemma shows that "weakest partial precondition" is not a useful concept, since $P1\{S\}Q \Leftrightarrow P2\{S\}Q$ does not imply that $P1\Leftrightarrow P2$, only that $t_s(a)\Rightarrow(P1\Leftrightarrow P2)$. This is the reason why the concept "strongest partial specification" is used here.

Using lemma 2 , we may easily prove the following rule (rules):

Rule 3 a) $\dfrac{P\{S\}Q}{((P\Rightarrow Q^x_{f_s(a)}) \wedge R^x_{f_s(a)})\{S\}R}$ b) $\dfrac{P\{S\}Q}{((P\Rightarrow Q^x_{f_s(a)}) \Rightarrow R^x_{f_s(a)})\{S\}R}$

Note that R can be any predicate independent of P and Q. This rule generalizes a given specification (P{S}Q) and will be called Rule of invocation. The term $P \Rightarrow Q^x_{f_s(a)}$ serves to make simplification of the precondition possible.

The precondition of the conclusion of b (where the $\wedge$ operator in a is replaced by a $\Rightarrow$ operator) is weaker. Still the two conclusions are equivalent.

If we want a precondition which is as "total" as possible, we should use rule 3 a.

Proof

3b) $R^x_{f_s(a)}$ {S} R ,by lemma 1

$(t_s(a) \Rightarrow R^x_{f_s(a)})$ {S} R ,by lemma 2

$((P \Rightarrow Q^x_{f_s(a)}) \Rightarrow R^x_{f_s(a)})$ {S} R ,by (1) and the partial function theorem of the premise

3a) follows from 3b) by (1) (stronger precondition) □

Note that we have proved (*) $R^x_{f_s(a)}$ {S} R ,
while in [5] the precondition of (*) had to be interpreted as $\forall y((x,y) \in f_s \Rightarrow R^x_y)$,or with our notation $t_s(a) \Rightarrow R^x_{f_s(a)}$.

The rule of invocation may be formulated without any occurrences of the function f_s as

Rule 3' $$\frac{P\{S\}Q}{(\forall v\ (P \Rightarrow Q^x_v) \Rightarrow R^x_v)\ \{S\}\ R}$$

This again follows from (2). However, this rule is weaker than rule 3 ,since the precondition here must hold for any v,not only $f_s(a)$.

This rule is presented both in [3] and [6], and also shown to be stronger than the rule of adaptation in [7].

Example:
Assume we have established a proof of

$\neg$full(q) {insert (q,x)} has (q,x)

where the variable q is a sequence of limited size and the variable x is of the element type. The predicates full and has, and the procedure insert should be self explaining.

In verification of a program calling insert, the postcondition ¬ has (q,x) may be required, i.e. we have the situation

$$?\{\text{insert } (q,x)\}\ \neg\text{has } (q,x)$$

By Rule of invocation, 3a, we get the precondition (denoting the function f_{insert} by f, the input variables are (q,x), the output variable is q):

$$(\neg\text{full}(q)\Rightarrow\text{has}(f(q,x),x)) \wedge \neg\text{has}(f(q,x),x)$$
$$\Leftrightarrow (\text{full}(q) \wedge \neg\text{has}(f(q,x),x))$$

Thus we have verified

$$(\text{full}(q)\wedge\neg\text{has}(f(q,x),x))\ \{\text{insert}(q,x)\}\neg\text{has}(q,x)$$

According to intuition full(q) must hold in the precondition.

If we use the rule of adaptation on this problem we get the precondition false. Now, false $\{S\}$ Q is valid for any S and Q, so this gives no information. Rule 3' gives the precondition $\forall v \neg\text{has}(v,x)$, which is false.

If we know that insert has no effect on q when q is full, say

$$\text{full}(q)\wedge q=q_o\ \{\text{insert}(q,x)\}\ q=q_o\ ,$$

then we may use rule 3b to obtain

$$((\text{full}(q)\Rightarrow q=f(q,x))\Rightarrow \neg\text{has}(f(q,x),x))\ \{\text{insert}(q,x)\}\neg\text{has}(q,x)$$
$$\Rightarrow \text{full}(q)\wedge\neg\text{has}(q,x)\ \{\text{insert}(q,x)\}\neg\text{has}(q,x)\ .$$

3. Effect functions

We define an effect function of S as an arbitrary total function which extends the unique partial function f_s, i.e. an effect function ,F_s, is a total function such that

(3) $\forall a \; t_s(a) \Rightarrow F_s(a) = f_s(a)$.

Note that for any S there will exist at least one such total function. If S always terminates normally, there is one unique effect function .

Thus, an effect function F_s satisfies

$\forall a \; F_s(a) =$ **if** $t_s(a)$ **then** $f_s(a)$ **else** $F(a)$ **fi**

where F is a given arbitrary total function.

In the following,total functions are denoted by capital letters. An effect function of S will be denoted by the subscript S.

It is sometimes convenient to extend the range of a total function by the value "undefined" ,for instance specifying division by zero as "undefined". Normal termination does not result in "undefined" :

(4) $t_s(a) \Rightarrow F_s(a) \neq$ undefined .

"Undefined" must satisfy

F(undefined)=undefined

and if v is a vector (or a list),

v_i=undefined $\Rightarrow$ v=undefined .

One may distinguish the two classes of "undefined" values : F(x)=undefined when x≠undefined,and F(undefined) . The former class may even be regarded as less serious than the latter (conf. lazy evaluation) .

To have a rich verification language, "undefined" can be refined into different values, for instance overflow and underflow, which correspond to different error situations [8].

Lemma 3

Let F_s be an effect function of S, then

(5) $R^x_{F_s(a)}$ {S} R

is the strongest partial specification of S given R.

Obviously, if an effect function of S is known, we may mechanically construct a precondition of S and R giving the strongest partial specification.

Proof

i) $R^x_{F_s(a)}\{S\}R$

$\Leftrightarrow$, by lemma 1

$\forall a\ t_s(a) \Rightarrow (R^x_{F_s(a)} \Rightarrow R^x_{f_s(a)})$

$\Leftrightarrow$ true , by (3)

ii) assume $P\{S\}R$

$\Leftrightarrow$, by reasoning as in i)

$\forall a\ (t_s(a) \wedge P) \Rightarrow R^x_{F_s(a)}$

we conclude

$$\left(R^x_{F_s(a)}\{S\}R\right) \Rightarrow \left((t_s(a) \wedge P)\ \{S\}R\right) \Leftrightarrow \left(P\{S\}R\right)$$

so (5) is the strongest partial specification given S and R. □

A partial specification can be used to determine whether a given function is an effect-function and also give a necessary condition for normal termination, by rule 2.

If the given partial specification is strong enough, the partial function theorem defines f_s completely within t_s, say we may prove something of the form

$$\forall a\ t_s(a) \Rightarrow (H_s(a) \wedge f_s(a) = F(a))$$

where the predicate $H_s(a)$ has no occurrence of f_s (nor t_s) and F is a given total function. F is an effect function of S by definition (3). $H_s(a)$ is a necessary condition for normal termination of S, since $\forall a\ t_s(a) \Rightarrow H_s(a)$.

If we like to use $H_s(a)$ as a guard of normal termination we may define another effect function

$$F'_s(a) = \underline{\text{if}}\ H_s(a) \underline{\text{then}}\ F(a)\ \underline{\text{else}}\ \text{undefined}\ \underline{\text{fi}}\ .$$

Now F'_s can be used to obtain necessary conditions for normal termination using (4).

Example.

Let S be the program

$\underline{\text{if}}\ m < 100\ \underline{\text{then}}\ m := m+1\ \underline{\text{else}}\ \text{error}\ \underline{\text{fi}}$

Take the postcondition $m=m_o$, where m_o is an auxiliary variable. We may generate the following precondition (using the Hoare System):

$m < 100 \Rightarrow m+1 = m_o$

Thus we have proved $(m < 100 \Rightarrow m_o = m+1)\{S\}\, m=m_o$. The partial function theorem is

$$\forall m\ \forall m_o\ t_s(m) \Rightarrow ((m<100 \Rightarrow m_o = m+1) \Rightarrow m_o = f_s(m))$$

$$\Downarrow$$

$$\forall m\quad t_s(m) \Rightarrow (m<100 \wedge f_s(m) = m+1)$$

The function F defined by $F(m)=m+1$ is an effect function of S. $m<100$ is a necessary condition for normal termination. The function F' defined by F'(m)= if $m<100$ then $m+1$ else undefined fi is also an effect function of S, which will guard against too large values of m.

An assignment to an array element, $a_i := x$, may terminate abnormally and can be interpreted informally as

(*) if $i \in I$ then $a_i := x$ else error fi ,

where I is the (limited) index range.

Define $(a|i|x)$ without index restrictions, by

$$(a|i|x)_j \equiv \text{if } i=j \text{ then } x \text{ else } a_j \text{ fi} \quad .$$

We may now prove that $(a|i|x)$ is an effect function of (*). Also if $i \in I$ then $(a|i|x)$ else undefined fi is an effect function of (*). This indicates that in partial verification reasoning, one may regard arrays as of infinite length . (A formal treatment of this is not given here.) □

A more general result obtained from a partial function theorem is:

(6) $\forall a\ t_s(a) \Rightarrow (C_{s,F}(a) \Rightarrow f_s(a) = F(a))$

$C_{s,F}(a)$ is a condition restricting the domain where the given function F can be used as an effect function of S. We will call F a conditional effect function of S (restricted by $C_{s,F}$).

This can be applied according to the rule

Rule 5

$$(C_{s,F}(a) \wedge R^x_{F(a)})\ \{S\}R \qquad ,C_{s,F} \text{ and F as in (6).}$$

The proof follows by lemma 1 . Hence we have found that a condition restricting a (conditional) effect function must be added in the precondition in order to use the conditional effect function .

In order to obtain a conditional effect function for S the mechanical postcondition $x=x_o$, where x_o is a vector of auxiliary variables, is convenient.
We will get a partial function theorem of the form

(7) $\forall a\ \forall x_o\ t_s(a) \Rightarrow (P(x_o,a) \Rightarrow f_s(a) = x_o)$,where $P(x_o,a)$ denotes the resulting precondition

We may prove

$$\forall a\ \ t_s(a) \Rightarrow (P(F(a),a) \Rightarrow F(a) = f_s(a)),\ (F \text{ arbitrary})$$

so any given function F is a conditional effect function of S. However, one should try to select F such that $P(F(a),a)$ is weak. Hopefully, the choice of F is obvious from P. If $P(F(a),a)$ is true then F is an (unconditional) effect function of S

Also a necessary condition for normal termination can be obtained from (7) :

$$\forall a\ t_s(a) \Rightarrow H_s(a) \quad \text{,if } \forall a\ \neg H_s(a) \Rightarrow P(x_o,a)\ .$$

Proof

$$(7) \Rightarrow \forall a,x_o\ \ t_s(a) \Rightarrow (\neg H_s(a) \Rightarrow f_s(a)=x_o) \quad \text{,if } \neg H_s(a) \Rightarrow P(x_o,a)$$
$$\Rightarrow \forall a\ \ t_s(a) \Rightarrow (\neg H_s(a) \Rightarrow \forall x_o\ f_s(a)=x_o)$$
$$\Rightarrow \forall a\ \ t_s(a) \Rightarrow H_s(a) \quad \text{,since} \forall x_o\ f_s(a)=x_o \text{ is false}\ .$$

Note that the rules introduced allow a strong postcondition (like $x=x_o$) even if the internal specification is weak. The precondition obtained will then be correspondingly strong.

Effect functions are most useful for treatment of procedure calls and loops. Both kinds of statements are specified isolated from the program text in which they appear (- procedures, since any program using them, is apriori unknown - loops since specification is done by invariants, which do not always fit into the surrounding programs). To handle procedures the given rules can be worked out to allow parameters as done in [4,7] .

4. Partial Interpretation of Data Types.

We shall now indicate how the results presented can be applied to data types [2,8,9,10]. In order to "verify a data type" total correctness is required. Here we define verification of a data type ,T, by"partial interpretation" which is within partial verification reasoning; and show that programs on the abstraction level of T can be partially verified. A formal treatment is beyond the scope of this paper.

A data type is defined, abstractly, by a set of functions and an axiomset on these; and can be regarded as a theory.

A concrete definition (an implementation) of a data type consists of i) a concrete representation of the abstract values (a vector of less abstract types) and ii) programs (bodies) implementing the abstract functions, operating on the concrete representation. An implementation can also be regarded as a theory [10].

Then, to verify (the implementation of) a data type means to prove that the concrete theory is an interpretation of the abstract theory. This is done by defining an interpretation,I, mapping an abstract term to its concrete representation, such that

$$I(G) = f_{G'} ,$$

where G' is the body of the concrete implementation of G. Now, to verify (the implementation of) a data type means to prove the interpretation of the abstract axioms in the concrete theory. This can not be done from partial function theorems alone since $t_{G'}$ is not known for any G'. Results based on total correctness are required.

In order to get some results based on partial correctness, we define an interpretation I', such that the only difference between I and I' is that $I'(G) = F_{G'}$, where $F_{G'}$ is an effect function of G'. To verify (the implementation of) a data type by partial interpretation means to prove the I'-interpretation of the abstract axioms in the concrete theory of effect functions.

In other words we prove that the I'- interpretation of the

abstract functions are effect functions of the concrete operations. Thus the abstract functions may be used as effect functions in programs on the abstraction level of the data type.

Let t be a variable of a data type T which is verified by partial interpretation. A call of an abstract function G in T may be represented by a statement t.G(a) which changes the concrete value of t to the function value of G'. We have

$$R^{t}_{G(t,a)}\{t.G(a)\}R$$

We can partially verify programs using data types, which are verified by partial interpretation.

If all the effect functions are given, then the theory of effect functions is consistent, and from verification by partial interpretation we can conclude that the abstract theory is consistent.

We may verify a data type T (totally) in two steps:

i) verify T by partial interpretation
ii) for each concrete body G' prove $f_{G'} = F_{G'}$ where $F_{G'}$ is the effect function used in i) .

Example.
A set-like type T_E, where E is the element type, can be defined as a data type which has the abstract functions Add: $T_E \times E \to T_E$, and Has: $T_E \times E \to$ Boolean (and other functions). For Add and Has we may have the abstract axiom:

Has(Add(q,x),y) = if x=y then true else Has(q,y) fi.

T_E may be implemented on the representation (a,m), where the variable m is of type [1:n] and a is an array [1:n] of E. (n is a constant). We define I(q) = (a,m).
Has may be implemented as a boolean function,has:

```
has(a,m,x) ≡
begin has:=false;
  for i:=1 to m do{invariant ∀j:[1:i-1] (a_j ≠ x)}
     if a_i=x then  has:=true; exit fi
end
```

Add may be implemented as a procedure, add, changing the representation:

add(a,m,x) ≡

if m<n then m:=m+1 ; a_m:=x else error fi

For has we can obtain the effect function H (and we define I'(Has(q,x)) = H(a,m,x)).

H(a,m,x) = $\exists j:[1:m](a_j=x)$

For add we can obtain the effect function A

A(a,m,x) = ((a|m+1|x),m+1)

((a|i|x) defined as on page 9)

To verify the implementation by partial interpretation we must prove

H(A(a,m,x),y)=if x=y then true else H(a,m,y)

which follows easily from the definition of A and H.

However, we can not (totally) verify the given implementation . In order to prove (by total correctness) f_{add} = A we must change A (to A') as in the example on page 8-9. If the abstract axiom is changed correspondingly, reflecting the capacity constraint , we may, using A', verify the implementation by partial interpretation; and together with a proof of f_{add}=A' and f_{has}=H we have verified the implementation (totally) .

Acknowledgements.

The author is greatly indebted to Ole-Johan Dahl for his constructive criticism.

The author would also like to thank Reiji Nakajima for many fruitful discussions.

REFERENCES

1. Hoare: An Axiomatic Basis for Computer Programming. Comm.ACM, 12,10 (Oct.1969).

2. Hoare: Proof of Correctness of Data Representations. Acta Informatica, 1, 1972.

3. Schjøll,Hesjedal, and Owe: PROVER Lecture Notes nr.24, Institute of Mathematics, University of Oslo, 1976.

4. Clint,Hoare: Program Proving: Jumps and Functions. Acta Informatica,1, 1972.

5. Ashcroft,Clint, and Hoare: Remarks on "Program Proving:Jumps and Functions by M.Clint and C.A.R. Hoare". Acta Informatica, 6, 1976.

6. Guttag,Horning, and London: A Proof Rule for Euclid Procedures. IFIP, Working Conference on the Formal Description of Programming Concepts, Aug. 1977.

7 Hoare: Procedures and Parameters: An Axiomatic Approach. Symp. on Semantics of Algorithmic Languages,Springer-Verlag, 7

8. Goguen: Abstract Errors for Abstract Data Types. IFIP, Working Conference on the Formal Description of Programming Concepts New Brunswick,N.J.Aug.1977

9. Burstall, Goguen: Putting Theories Together to make Specifications. Int.Jt.Conf. on A.I., 1977.

10. Nakajima,Honda, Nakahara: Describing and Verifying Programs with Abstract Data Types. in "Formal Description of Programming Concepts". North Holland Publ. Comp. 1977.